Name → Anamika
Section → A

Subject → Design and Analysis of Algorithm
Roll no. → 2014559

Ans 1. Asymptotic notations are the mathematical notations used to describe running time of an algorithm when the input tends towards infinity value. (particular limiting value). These notations are generally used to determine the running time of an algorithm and how it grows with the amount of input.

There are 5 types of asymptotic notations:

(i) Big Oh (O) notations → They define an upper bound of an algorithm it bounds the function only from above formally:

$$O(g(n)) = \{f(n) : \text{there exists positive constant } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \; \forall \; n \ge n_0\}$$

(ii) Small Oh (o) notation: We use o-notation to denote an upper bound that is not asymptotically tight.

formally: $o(g(n)) = \{ f(n) :$ for any positive constant $c > 0$ there exists a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n) \quad \forall \ n \geq n_0 \}$

(iii) Big omega (Ω) notation: It denotes asymptotic lower bound more formally:

$\Omega(g(n)) = \{ f(n) :$ for any +ve constant c and n_0 $0 \leq cg(n) \leq f(n) \quad \forall \ n \geq n_0 \}$

(iv) Small omega (ω) notation: By analogy, ω notation is used to Ω notation as o-notation is to O-notation. We use ω-notation to denote a lower bound that is not asymptotically tight. Formally:

$\omega(g(n)) = \{ f(n) :$ for any +ve $c > 0$, there exists $n_0 > 0$ such that $0 \leq cg(n) < f(n) \quad \forall \ n \geq n_0 \}$

(v) Theta (θ) notation: The theta notation bound the func. from above and below so it defines exact asymptotic behaviour. Formally:

$\theta(g(n)) = \{ f(n) :$ there exists +ve constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall \ n \geq n_1 \}$

Ans 2. $O(n)$

Ans 3. $T(n) = \begin{cases} 3T(n-1) & \text{if } n>0 \\ 1. \text{ else} & n \le 0 \end{cases}$

By using backward sub. →

$T(n) = 3T(n-1)$ —①

$T(n-1) = 3T(n-2)$ —②

$T(n-2) = 3T(n-3)$ —③

Putting eqn ② and ③ in ① →

$T(n) = 3(3T(n-2))$

$T(n) = 3 \times 3 \times 3T(n-3)$

$T(n) = 3^3 T(n-3)$

In general → $T(n) = 3^k T(n-k)$

Let $n-k=0$ for base case so, $n=k$

⇒ $T(n) = 3^n T(n-n)$

$= 3^n T(0) = 3^n$

So, Time complexity → $O(3^n)$

Ans 4 ⇒ $T(n) = \begin{cases} 2T(n-1)-1 & n>0 \\ 1 & n \le 0 \end{cases}$

By using Backward Sub →

$T(n) = 2T(n-1)-1$ —①

$T(n-1) = 2T(n-2)-1$ —②

$T(n-2) = 2T(n-3)-1$ —③

Putting ② & ③ in ① →

$$T(n) = 2 * 2T(n-2) - 2 - 1$$

$$T(n) = 2 * 2 * 2T(n-3) - 4 - 2 - 1$$

$$T(n) = 2^3 T(n-3) - 4 - 2 - 1$$

In general → $T(n) = 2^k T(n-k) - 2^{k-1} - 2^{k-2} \ldots 2 - 1$

For base case $n-k = 1 \Rightarrow n = k$

So, $T(n) = 2^n T(n-n) - 2^{n-1} - 2^{n-2} \ldots - 2 - 1$

$$\Rightarrow 2^n - 2^{n-1} - 2^{n-2} \ldots - 2 - 1$$

By Taking the bigest term $T(n) = 2^n$

So, Time compleity is $O(2^n)$

Ans 5: $O(n)$

Ans 6: $O(\sqrt{n})$

Ans. 7: $O(n \log n \log n)$

Ans 8: The re occurrence relation is:

$$T(n) = \begin{cases} T(n-3) + n^2, & n > 1 \\ 0, & n \leq 1 \end{cases}$$

by using backward substitution: —

$$T(n) = T(n-3) + n^2 \quad -①$$

$$T(n-3) = T(n-6) + (n-3)^2 \quad -②$$

$$T(n-6) = T(n-9) + (n-6)^2 \quad -③$$

Putting ② and ③ in ① →

$$T(n) = T(n-6) + (n-3)^2 + n^2$$

$$T(n) = T(n-9) + (n-6)^2 + (n-3)^2 + n^2$$

In general, $T(n) = T(n-3k) + (n-3(k-1))^2 + (n-3(k-2))^2 + \dots n^2$

For base case → $n - 3k = 0 \Rightarrow k = n/3$

So, $T(n) = T(n-n) + (n-3(\frac{n}{3}-1))^2 + (n-3(\frac{n}{3}-2))^2 + \dots n^2$

$$= 0 + n^2 + (n-3)^2 + (n-6)^2 \dots + (n-3(\tfrac{n}{3}-1))^2$$

Taking only higher order terms we get →

$T(n) = (n^2 + n^2 + \dots n^2)$ + (other terms) → ignored since lower order

~~So, Time complexity → $O(n^2)$.~~

$n/3$ times

So, $T(n) = \frac{n}{3} \times n^2 = \frac{n^3}{3}$

So, Time complexity $O(n^3)$

Ans 9: We can get sum series as :- → (for log)

$$S = n + \frac{n}{2} + \frac{n}{3} + \dots \frac{n}{n}$$

$$S = n\left(1 + \frac{1}{2} + \frac{1}{3} + \dots \frac{1}{n}\right)$$

$$S = n \log n$$

So, Time complexity is $O(n \log n)$

Ans 10: $f(n) = n^k \quad k \geq 1$

$g(n) = a^n \quad a > 1$

Exponential functions grow faster than polynomial functions hence:

$$O(n^k) < O(a^n)$$

for values of $k \geq x$ and $a \geq y$

Let calculate x and y

Let $k = 2$ and $a = 2$ as well.

$$f(n) = n^2 \;, \; g(n) = 2^n$$

take log on both sides →

$\log(f(n))$	$\log(g(n))$
$= 2 \log_2 n$	$= n \log_2 2 = n$
$\simeq \log n$	

So, $O(\log n) < O(n)$

Hence for $k \geq 2$ and $a \geq 2$ the condition satisfies.

Ans 11: Here the value of i goes as →

1, 3, 6, 10, 15, ----

And the sum of a series $1+2+3+$ ---upto n is $\frac{n(n+1)}{2}$

So, the above series for i will stop where an become equals to or greater than n hence →

$$\frac{n(n+1)}{2} = n_0$$

$$\Rightarrow n^2 + n = 2n_0$$

$$\Rightarrow n^2 + n - 2n_0 \geq 0 \quad \Rightarrow n = \frac{-1 \pm \sqrt{1 - 8n_0}}{2}$$

So, $n \simeq \sqrt{n_0}$

Hence, Time complexity is $O(\sqrt{n})$

in worst case it will be $O(n)$ for the recursion stack which go to n

Ans 12: $T(n) = \begin{cases} T(n-1) + T(n-2) + 1 & n \geq 2 \\ 1 & 0 \leq n < 2 \end{cases}$

Let $T(n-2)$ takes time app. equal to $T(n-1)$

So, $T(n) = 2T(n-2) + C$ ① (same constant)

$T(n-2) = 2T(n-4) + 2C$ ②

$T(n-4) = 2T(n-6) + 3C$ ③

Using all ③ eqns →

$T(n) = 2*2*2T(n-2\times3) + 3C + 2C + C$

in general → $T(n) = 2^k T(n-2k) + (1+2+3\ldots k)C$

for base case → $n-2k = 1 \Rightarrow k = \frac{(n-1)}{2}$;

$$T(n) = 2^{\frac{(n-1)}{2}} T(1) + \frac{k(k+1)}{2} C$$

$$T(n) = 2^{\frac{(n-1)}{2}} T(1) + \frac{n(n-1)}{4} C \quad \left[\because k = \frac{n-1}{2}\right]$$

$$T(n) \approx 2^{\frac{n}{2}} + \frac{n(n-1)}{4} C$$

So, Time complexity is $O(2^n)$

13 (i) with $T.C = O(n \log n)$

Pgm:

```
for (i=0; i<n; i++)
{
    for (j=0; j<n; j*=2)
    {
        printf("x");
    }
}
```

(iii) with T.C. = $O(n^3)$

```
for (int i=0; i<n ; i++)
  {
    for (int j=0; j<n ; j++)
      {
        for (int k=0; k<n; k++)
          cout << "X" << "Y";
      }
  }
```

(iv) with T.C. = $O(\log(\log n))$

```
int func (int n)
{
  int c=0;
  while (n>0)
  {
    c++;
    n/=2;
  }
  return c;
}

int x = func(n);
for (int i=1; i<=x; i*=2)
  {
    cout << "hi";
  }
```

Ans 14 - $T(n) = T(\frac{n}{4}) + T(\frac{n}{2}) + cn^2$

we can assume $T(\frac{n}{2}) \geq T(\frac{n}{4})$

So, $T(n) = 2T(\frac{n}{2}) + cn^2$ — (1)

Using master's theorem and comparing with standard equation $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

$a = 2, \quad b = 2$

So, $c = \log_b a = \log_2 2 = 1$

$\Rightarrow$ and $f(n) = n^k$

So, $cn^2 = n^k$

$\Rightarrow n^2 = n^k$

$\Rightarrow k = 2$

$\therefore f(n) > c \Rightarrow 2 > 1$

Complexity $= O(f(n)) = O(n^2)$

Ans 15: Since the series is ~~list~~ going as $\rightarrow$

$$S \Rightarrow n + \frac{n}{2} + \frac{n}{3} + \frac{n}{4} \cdots \frac{n}{n}$$

$$S = n\left(1 + \frac{1}{2} + \frac{1}{3} + \cdots \frac{1}{n}\right)$$

$$S = n \log n$$

So, $T.C = O(n \log n)$

Ans 16: Assuming pacv (i,k) works in $\log(k)$ time we can express the runtime as $\rightarrow$

$$\sqrt[k]{n^{\sqrt[k]{\cdots \sqrt[k]{n}}}} \leq \sqrt[k]{2}$$

$$n^{\frac{1}{k^m}} \leq 2^{\frac{1}{k}}$$

raise both side to k^m

$$\Rightarrow n \leq 2^{k^{m-1}}$$

Take log $\log(n) \leq k^{m-1} \log_2 2$

Take log once again $\Rightarrow \log(\log(n)) \leq (m-1)\log k$

$\log(\log(n)) + 1 \leq m$

$\because$ pow(i, k) takes $\log(k)$ time

Complexity $\Rightarrow O(\log(k) \cdot \log(\log(n)))$

Ans 18: a) $100 < \log(\log(n)) < \log(n) < \sqrt{n} < n < \log(n!)$
$< n\log n < n^2 < 2^n < 2^{2n} < 4^n < n!$

b) $1 < n < 2n < 4n < \log(\log n) < \log(\sqrt{n}) < \log(n)$
$< \log(2n) < 2\log(n) < \log(n!) < n\log n < n^2 < (2^n)2 < n!$

c) $96 < \log_8(n) < \log_2(n) < \log_6 n < n\log_2 n < \log(n!) < 5n$
$< 8n^2 < 7n^3 < 8^{2n} < n!$

Ans 19:
```
for (int i=0; i<n; i++)
  {
    if (arr[i] is equal to key)
        print index and break.
    else
        continue
  }
```

Ans 20: Iterative:
```
void insertionSort (vector<int>& arr)
{ int n = arr.size();
  for (int i=0; i<n; i++)
  { int j = i;
    while ( j > 0 and arr[j] < arr[j-1])
    {
      swap (arr[j], arr[j-1]);
      j--;
    }
  }
}
```

Rewrisive :

```
void insertionSort (vector<int>& arr, int i)
{
    if (i<=0) return;
    insertionSort (arr, i-1);
    int j=i;
    while (j>0 & arr[j]<arr[j-1])
    {
      swap (arr[j], arr[j-1]);
      j--;
    }
}
```

It is called online sorting algorithm because it does not have the constraint of having the entire input available at the beginning like other sorting algorithms as bubble sort or selection sort. It can handle data piece by piece.

Ans 21) Quick Sort : $O(n \log n)$
Merge Sort : $O(n \log n)$
Bubble Sort : $O(n^2)$
selection Sort : $O(n^2)$
Insertion Sort : $O(n^2)$

Ans 22 : Inplace : Bubble, Selection, Quick, Insertion Sorts.
Stable : Bubble, Insertion, Merge Sorts.
Online : Insertionsort

Ans 23) Iterative:

```
int low=0, high=n-1;
while (low <= high)
{
    mid = (low+high)/2;
    if (key == a[mid])
        print mid and break;
    if (key > a[mid]) low = mid+1;
    else high = mid-1;
```

Recursive:

```
int BS (arr, low, high, key)
{
    if (low > high) return -1;
    mid = (low+high)/2;
    if (arr[mid] == key] return mid;
    if (arr[mid] > key] return BS (arr, low, mid-1);
    else return BS (arr, mid+1, high);
}
```

Time Complexity of BS:
- Iterative = O(logn)
- Recursive = O(logn)

Space Complexity of BS:
- Iterative : O(1)
- Recursive: O(logn)

Time Complexity of Linear Search:
- Iterative = O(n)
- Recursive = O(n)

Space complexity of Linear Search:
- Iterative = O(1)
- Recursive = O(n)

24) Recurrence relation for B.S.:

$$T(n) = T\left(\frac{n}{2}\right) + 1$$